Extending ARM for End-to-End Correlation and Optimization of E-service Transactions

Akhil Sahai, Jinsong Ouyang, Vijay Machiraju
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-168
December 11th, 2000*

optimization, management, E-services, ARM

E-services on the web are increasingly beginning to cooperate to perform complex tasks. This has resulted in transactions that although begin and end at an e-service have parts of them executed on other cooperating e-services. Because of their federated nature and varied implementation, it is difficult to perform end to end correlation of such transactions that span multiple e-services. The problem would be compounded by the increasing trend of dynamic composition of e-services. Application Response Measurement is a standard for measuring application responses that originate at a client and involve multiple servers. We extend ARM to the e-services world to perform end to end correlation. In doing so we enable multi-dimensional correlation-a local correlation as well as a correlation spanning multiple e-services.

* Internal Accession Date Only
Approved for External Publication

Extending ARM for End-to-End Correlation and Optimization of E-service Transactions

Akhil Sahai, Jinsong Ouyang, Vijay Machiraju
HP Laboratories, 1501 Page Mill Road, Palo-Alto, CA 94034

Abstract: E-services on the web are increasingly beginning to cooperate to perform complex tasks. This has resulted in transactions that although begin and end at an e-service have parts of them executed on other cooperating e-services. Because of their federated nature and varied implementation, it is difficult to perform end to end correlation of such transactions that span multiple e-services. The problem would be compounded by the increasing trend of dynamic composition of e-services. Application Response Measurement is a standard for measuring application responses that originate at a client and involve multiple servers. We extend ARM to the e-services world to perform end to end correlation. In doing so we enable multi-dimensional correlation-a local correlation as well as correlation spanning multiple e-services.

A. INTRODUCTION

The term E-service has been already overloaded. An *e-service* can be described broadly as a service available via the Internet that conducts transactions. E-businesses set up E-Services for clients and other E-Services to access. They have a Uniform Resource Locator at which they can be accessed and have a set of Interfaces that can be utilized to access them. E-services that are capable of intelligent interaction would be able to discover and negotiate with each other, mediate on behalf of their users and compose themselves into more complex services. This composition could be static or dynamic. These E-services by virtue of their deployment by different enterprises cross enterprise boundaries during their interactions. The interaction thus happens across management domains and enterprise networks. For example, *priceline.com* and *expedia.com* act as the broker for airlines, hotel and car booking respectively. They are statically composed e-services that have pre-negotiated understanding with certain airlines and hotel services and broker their services through their portal sites. In the dynamic composition scenario, which is increasingly the trend, these e-services would enter into agreements on the fly and compose themselves to serve a set of requests by discovering each other at E-MarketPlaces (Brokers).

As these e-services are being deployed by different enterprises and because of usage of different technologies for their implementations, these e-services could be vastly different in nature. They could be based on CORBA [4], BizTalk [3], COM, J2EE [10], E-speak [2] and they would need to agree upon document exchange protocols to communicate and inter-operate with each other.

The need for end-to-end correlation of transactions spanning multiple e-services arises from the perspective of both clients and service providers. Clients are interested in tracking their interactions and understanding the e-service process flow internals. This enables the client to seek some insight to the actual e-service flow. For example, FedEx provides its clients information about their packet location as it traverses its itinerary.

E-service providers that are using other e-services to provide composite services would like to know how the component services are behaving. By studying and observing their behavior a composite e-service would be able to optimize itself by either changing its component sub e-services or by instructing the existing component sub e-services to improve performance.

ARM [7] is a standard in intrusive application instrumentation. It provides for correlation of application level transactions. As our approach extends ARM to e-service transactions it is necessary to understand ARM functionality.

B. ARM OVERVIEW

ARM provides APIs to delimit sections of application code base. A section of an application code base can be made into a *transaction* by inserting start and stop calls. Handles can be attached to such transactions. The local ARM Agent maintains the response time details of such transactions. ARM also provides for transactions that span multiple servers

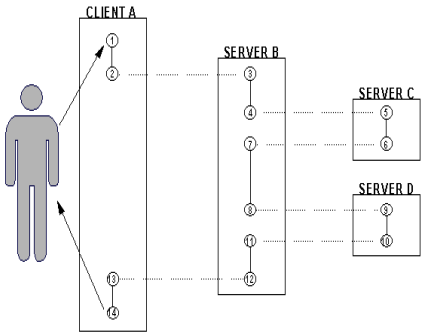

Figure 1: ARM can be used to delimit transactions

ARM 2.0 API adds interesting functionality in the form of correlation and furnishing of application data in the form of data buffers, which are maintained by the ARM library. The data buffers can contain application specific data. The correlation data is sent to a central correlation application that undertakes the task of linking up the transactions with their component transactions.

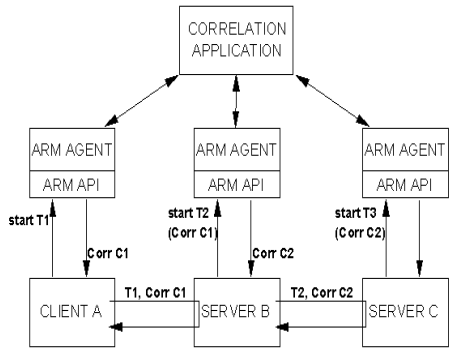

Figure 2: Correlation in ARM

C. INADEQUACY OF ARM IN THE E-SERVICES WORLD?

- ARM provides only lateral correlation.

E-services have their own workflow engines. Depending on which state the e-service is in; a transaction is initiated on request from one or more remote e-service(s)/client(s). Thus for a transaction there could be multiple parents, a local parent corresponding to the local work flow engine and others corresponding to the remote e-service transactions. ARM does not have the capability of doing this kind of multi-dimensional correlation.

- ARM assumes a central correlation engine.

In the E-service world this central correlation engine has to be replaced by a management system equally trusted by the e-services participating in the management task [9]. As the management data needs to be sent over the Internet and the number of transactions being created are so large that they can easily overwhelm a simple correlation engine, local caching and selective transfer of data needs to be done.

- ARM assumes NOM (Network Object Model) mechanism for correlation.

E-services essentially use the Document Object Model of interaction. Correlators need to be sent and retrieved using Documents from an e-service to another.

- ARM only provides response time details by default

For analyzing and optimizing e-services a host of data is required which ARM does not provide. ARM provides for application specific data to be collected but there is no agreed upon format for this data.

D. EXTENDING ARM FOR CORRELATING E-SERVICES DATA

At any given instant of time the e-services form a tree or a directed acyclic graph with e-services at various nodes (refer figure 1). As these DAGs are created either statically or dynamically and conversations/transactions are performed management information are sent to the E-Service Manager, which performs correlation, analysis and management. To enable end-to-end correlation and management of e-service transactions, we extend the ARM protocol and term it as the XARM (eXtended ARM) protocol, which is used to exchange the XARM objects (transaction traces) between the managed e-services and the central e-service manager. These XARM objects contain two types of information at every level: correlators and management information objects (MIO). As an e-service conversation can go across multiple e-services, the correlation information (correlator) is needed to track a conversation and correlate the management data collected at each participating e-service. As the conversations are performed the MIOs would be correlated.

E-services at each level can further enrich the data collection by agreeing on a certain ESSMIO (E-service specific MIO) defined in a particular schema that would enable the collaborating e-services to furnish business logic data (e.g. the number of book buy requests received etc). The default XARM object contains the correlator and a certain set of predefined management information data as described in section G.

1st. XARM protocol

The XARM API and its library maintain a local picture of a transaction. This correlates the local parent, itself and the local children transactions. For each transaction, the XARM library generates a correlator and an MIO and sends the correlator to the central e-service manager at the start of the transaction. The library maintains and updates the transaction's MIO locally during the lifecycle of the transaction, and sends it to the e-service manager at the end of the transaction.

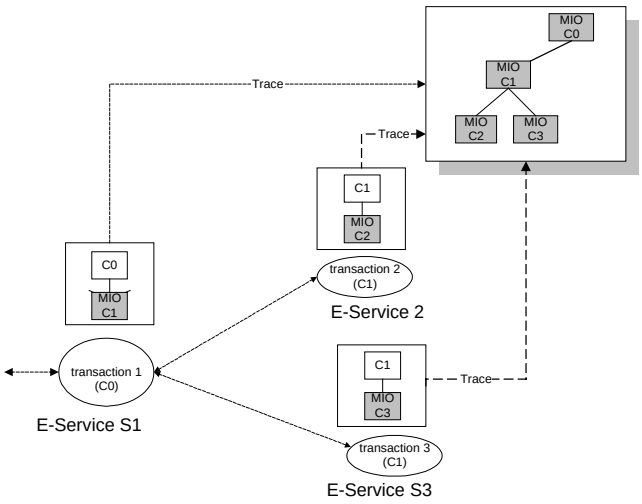

Figure 3: XARM protocol

Figure 3 shows the workflow of the XARM protocol. A transaction, at its start, generates a context by combining its own correlator and that from its parent (if any), and sends the context to the e-service manager. The e-service manager then adds the transaction to the children list of its parent. At the end of a transaction, the MIO containing the management statistics is sent to the e-service manager. Thus as shown in figure 3, a partial tree is drawn when e-service s_1 ends its transaction and sends a response back to its client, e-service s_0. New children would be added to s_0 if it has more sub transactions.

2nd. XARM APIs

An e-service conversation/transaction usually starts with a request from a consumer or a parent e-service, and ends with a response or a new request to another e-service. In between, one or more of the following can occur, depending on how a conversation is defined and implemented.

- The e-service can reply with one or more preliminary responses.
- The e-service can receive one or more subsequent requests. A sub transaction can start with or without a subsequent request.
- The e-service can send one or more requests to one or more of its sub/external e-services.
- The e-service can receive one or more response from its sub/external e-services.

To manage e-service transactions with the above communication pattern, several issues must be addressed. First, the communication between e-services is asymmetric. That is, one or more response can correspond to one request, or vice versa. Furthermore, an e-service can send a response directly to the ultimate requester, instead of its parent. Second, a transaction is usually nested in both a local transaction hierarchy and a conversation across multiple e-services. Third, the request/response documents to be sent/received by an e-service can be handled by different threads/components. To address these issues for end-to-end correlation and management of e-service transactions, we propose the XARM API and present the details in the following.

Two types of information are collected from e-services. One type is *descriptive* information, which contains the e-service and transaction definitions. The other is *management* information, which contains the correlator and the MIO for each transaction. The schemas for correlator and MIO will be described in the following section.

To send transaction traces to the central management e-service, the XARM library needs to know the URI of the e-service manager, which is provided in the XARMInit class. An e-service, when getting started, initiates an instance of this class and calls XARM_configure to let the XARM library know the destination of the e-service manager.

```
public class XARMInit extends Object {
// Public Constructors
        public   XARMInit();
// Public Instance Methods
        public short   XARM_configure(
                        String  service_manager_URI,
                         int      flags);
// misc …
}
```

Descriptive information is provided in the class XARMTranRegistration. An e-service, when getting started, initiates an instance of this class and calls registration method, XARM_register_transaction, to define the mapping from a universally unique transaction class ID to a name pair (service, transaction). There are two versions of XARM_register_transaction, depending on whether the transaction ID is provided as a parameter, or generated by the XARM library. The library maintains a list of locally registered transactions. When method XARM_register_transaction is called, a new entry will be added in the list, and a registration trace is sent to the e-service manager.

```
public class XARMTranRegistration extends Object {
// Public Constructors
```

```
        public   XARMTranRegistration();
// Public Instance Methods
        public short   XARM_register_transaction(
                            String  service_URI,
                            String  tran_name,
                            byte[]  tran_id;
                            int     flags);
        public byte[]  XARM_register_transaction(
                            String  service_URI,
                            String  tran_name,
                            int     flags);
// misc …
}
```

Management information is provided in the class XARMServiceTran. This class represents e-service transactions when they execute. An e-service creates as many as instances it needs. This would typically be at least as many as the number of transactions that can be executing simultaneously. An e-service would typically create a pool of XARMServiceTran objects, take one from the pool to use when a transaction starts, and put it back in the pool after the transaction ends for later reuse. Internally, each entry in the list of registered transaction has a list of transaction instances. Each entry contains an XARM object of a specific transaction instance. When a new instance of a transaction class is started, a new entry will be allocated and added in the corresponding instance list. The contained XARM object will be updated at each stage of the transaction, and will be sent to the remote e-service manager at the end of the transaction. The maximum length of each transaction instance list depends on the system configuration. The definition of the class is as follows.

```
public class XARMServiceTran extends Object {
// Public Constructores
  public XARMServiceTran();
// Public Instance Methods

  public byte[] XARM_start_transaction
                (byte[] tran_id,
                 byte[][] parent_tran_handles,
                 ESSMIO essmi_object,
                 int flags);
  public short XARM_start_transaction
                (byte[] tran_id,
                 byte[][] parent_tran_handles,
                 byte[] tran_handle,
                 ESSMIO essmi_object,
                 int flags);
  public short  XARM_update_transaction
                (byte[] tran_handle,
                 ESSMIO  mi_object,
                 int flags);
  public short  XARM_stop_transaction
                (byte[] tran_handle,
                 int tran_status,
                 String description,
                 ESSMIO essmi_object,
                 int flags);
  public short XARM_complete_transaction
                (byte[] tran_id,
                 byte[][] parent_tran_handles,
                 byte[] tran_handle,
                 int tran_status,
                 Time resp_time,
                 ESSMIO essmi_object,
                 int flags);
// misc …
}

public class XARMUtil {

public  Document XARM_insert
                (Document document,
                 int doc_type,
                 byte[] tran_handle,
                 int flags);
  public byte[] XARM_retrieve
                (Document document,
                 int doc_type);
}
```

Method XARM_start is used to start a transaction. There are two versions of XARM_start, depending whether a transaction handle is provided by the user or generated by the XARM library. The other parameters to this method are the transaction class ID, the parent transaction handles, and the ESSMIO. The transaction ID is the type of transaction this transaction instance (the transaction handle) belongs to. The local parent transaction handle associates this transaction with its local parent if any. If the transaction is due to an external service request, method XARM_retrieve (a utility function) is called to retrieve the handle, part of the correlator tagged with the request. The handle is passed as a remote parent transaction handle. It associates this transaction with the remote transaction sending the request. The ESSMIO is used to contain the necessary e-service specific management information for this transaction, if any.

An e-service may need to send requests to its sub e-services during a transaction. In this case, it calls method XARM_insert (a utility function) to piggyback

the transactions correlator on the document to be sent. The correlator for a transaction consists of the transaction class ID and the transaction handle.

When XARM_start is called, the XARM library generates an XARM object for the transaction, and sends the object (a start trace) to the e-service manager. The XARM object is composed of the following.

- The correlators of the transaction and its local and remote parents if any
- The transaction's MIO
- The transaction's ESSMIO

Like ARM, an e-service transaction can update its statistics during its lifecycle by calling XARM_update. The XARM library will then update and send, in an update trace, the MIO and ESSMIO to the e-service manager.

XARM_stop is used to inform the XARM library the end of a transaction, and summarizes its statistics. It informs the XARM library the status of this completed transaction. The *description* can be optionally used to inform the management system some more details about the transaction. If the application logic finishes successfully, the status is set to XARM_GOOD. If the service request is not satisfied (e.g., could not reserve a hotel room due to no vacancy), the status is set to XARM_ABORTED. The status is set to XARM_FAILED if there is an application or system failure (e.g., a sub service is unavailable). The description in this case can detail the reasons of failure. XARM_stop will send, in a stop trace, the updated MIO and ESSMIO to the e-service manager.

Like ARM, XARM also provides method XARM_complete_transaction for e-services to pass the statistics for completed transactions. The method, based on passed parameters, builds an XARM object, and sends it to the e-service manager.

Figure 4 is an example that illustrates how the XARM API is used to instrument e-services. In the example, the retailer e-service sends a request to the delivery e-service, which then delivers the merchandise to the customer. There are two transaction classes in the delivery e-service: "delivery" and "items". The delivery transaction starts at a certain point of time each day, and stops at another time or after a certain number of merchandise deliveries. Each particular delivery is referred to as an instance of the item transaction. This is a typical example where XARM's multi-dimensional correlation is needed for e-service transaction management.

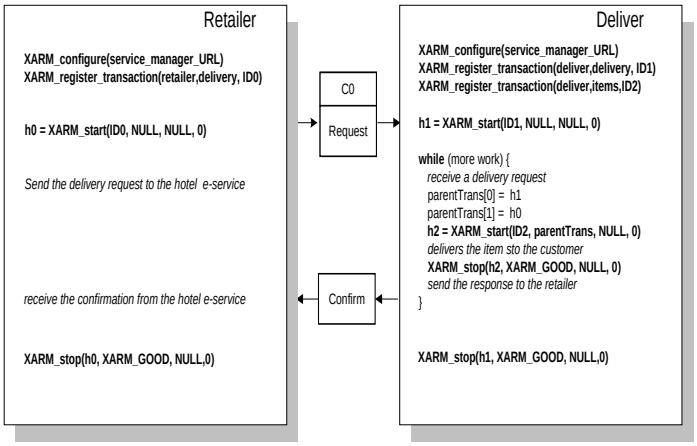

Figure 4: Usage of XARM APIs in a simple scenario

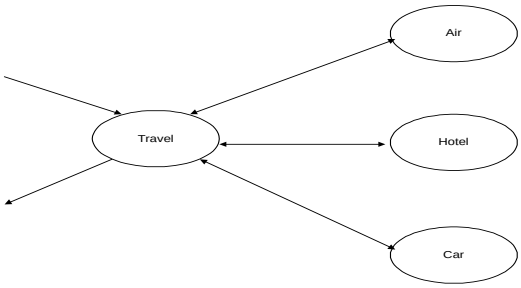

Figure 5: A travel e-service depends on Airline, car and hotel e-services

Another example would be a travel e-service that interacts with air flight, hotel, and car rental e-services. The interactions between these e-services are shown in figure 5, and figure 6 presents the instrumented pseudo code.

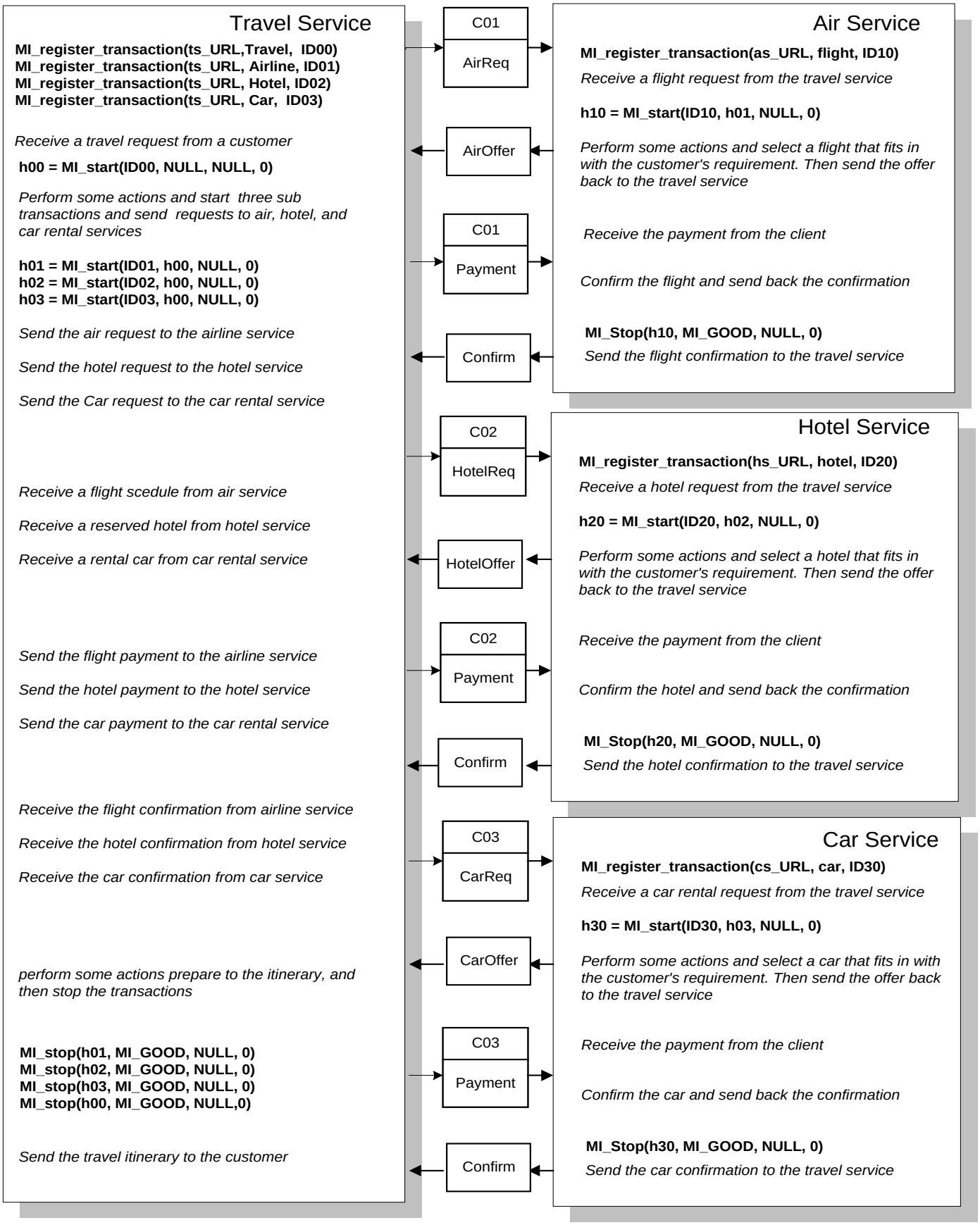

Figure 6: Instrumentation of travel e-service

E. XARM OBJECT FORMAT DEFINITION

An XARM object may contain some or all of the following: the correlators, a management information object (MIO), and an e-service specific MIO (ESSMIO), as shown figure 7. We have described the format of the correlator. In this section, we derive a set of management parameters [6] that should be included in the MIO, then present the XARM object schema [1].

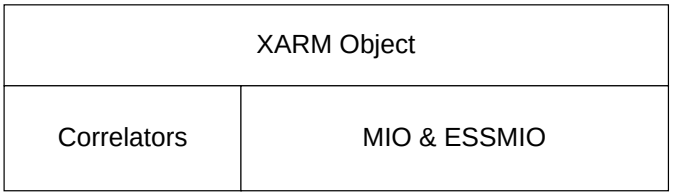

Figure 7. The XARM object

The management parameters in MIOs are classified into the following types.

- E-service health index. This indicates the current status of an e-service. Possible values are: *up, down, congested, halted, restarting,* and *unknown.*
- Availability. This indicates an e-service's availability, in terms of downtime per a certain period of time. Downtime is the duration of e-service unavailability due to system or application faults.
- Reliability. The reliability of an e-service can be measured in terms of the fault rate. That is, the number of faults per the number of handled service requests.
- Performance. Performance parameters include response time, response status, throughput/rate, aborted count/rate, and failed count/rate. Response time is the duration from the sending of a request to the receiving of a response. Response status indicates if a service request has been done, aborted, or failed. Throughput/aborted/failed rate is the number of committed/aborted/failed services per the total number of service requests.
- Faults. A fault parameter contains the information about a fault when a request is serviced (e.g., fault ID, fault description, time of the fault occurred).

The above parameters form a base for the management of e-service transactions.

By aggregating and analyzing MIOs for cooperating e-service transactions, the central management e-service can provide the necessary information to perform the following management tasks. It enables the management system to perform end-to-end conversation management. It enables an e-service or a consumer to track its interactions with other e-services and the e-service flow. It enables service ranking and selection. An e-service can rank its external e-service providers based on their performance, availability, and reliability. The latest statistics of the external e-services can also help the e-service choose the right e-service supplier. This kind of correlation leads to service optimization as well. By identifying performance bottlenecks and service failure points, the management agent can reconfigure and/or restart the e-service to achieve better performance and availability. This correlation mechanisms leads to e-service evolution. An e-service provider usually provides different types of services. The information provided by the XARM library and aggregated by the e-service manager can be used to monitor the access patterns for the provided services, and help make decisions about whether or how to evolve the non-performing services to generate more revenue. In addition it leads to consumer tracking. The information can be used to find out the consumer pattern so that actions can be taken to guarantee the QoS for valued customers.

3rd. The XARM object schema

Each XARM object contains the correlators of the transaction, and its local and remote parents. The correlator is transparently handled by the XARM library. Its schema is:

```
<complexType name = "CorrelatorType">
  <element   name = "TranID" type = "string"/>
  <element   name = "TranHandle"
             type = "String"/>
</complexType>
```

Currently the measurement data in each MIO contain the identity (i.e., URI for a consumer or service provider), and performance, availability, and reliability statistics.

```
<complexType name = "MIOType">
  <element name = "Identity" type = "IDType"/>
  <element name = "Performance"
           type = "PerfType"/>
  <element name = "Availability"
           type = "AvailType"/>
  <element name = "Failure"
           type = "FaultType"/>
</complexType>

<complexType name = "IDType">
  <element name = "From" type = "serviceURI"/>
  <element name = "to"   type = "serviceURI"/>
</complexType>
```

```
<complexType name = “PerfType”>
  <element name = “RespTime” type=“decimal”/>
  <element name = “StartTime” type=“decimal”/>
  <element name = “StopTime”  type=“decimal”/>
  <element name = “TranStatus”
           type = “decimal”/>
</complexType>

<complexType name = “AvailType”>
  <element name = “ReqCount” type =“decimal”/>
  <element name = “ComCount” type =“decimal”/>
  <element name = “FailCount” type=“decimal”/>
  <element name = “AbortCount”
           type = “decimal”/>
</complexType>

<complexType name = “FaultType”>
  <element name = “FaultId” type =“decimal”/>
  <element name = “FaultDesc” type =“string”/>
</complexType>
```

The Correlators and MIO construct a default XARM object that is used to identify a transaction’s local context and statistics.

```
<complexType name = “XARMType” >
  <element name = “Parent”
           type = “CorrelatorType”
           minOccurs = “0”
           maxOccurs = “unbounded”>
  <element name = “Correlator”
           type = “CorrelatorType”/>
  <element name = “MIO” type=“MIOType”/>
</complexType>
```

To identify a service request in the context of a conversation, an e-service needs to get the XARM objects of its predecessors. To know how its sub e-services perform, an e-service needs to get the XARM objects of the participating e-services. XARMTree is defined to serve the need. This is the structure generated by the remote e-service manager when receiving XARM objects from local XARM libraries.

```
<element name = “XARMTree”
         type =“XARMTreeType”/>

<complexType name = “XARMTreeType” >
  <element name = “Predecessor”
           type = “XARMType”
      minOccurs = “0“ maxOccurs = “unbounded”>
  <element name = “XARMObject”
           type = “XARMType”/>
  <element name = “Child”
           type = “XARMType”
      minOccurs = “0“ maxOccurs = “unbounded”>
</complexType>
```

F. CONCLUSION

End-to-end e-service transaction management is a challenging task. Although ARM addresses some of the issues it is inadequate in the domain of e-service transaction management. XARM extends ARM to this domain.

G. ACKNOWLEDGEMENTS

We would like to thank Aad Van Moorsel and Klaus Wurster for his suggestions and feedback on the paper.

H. REFERENCES

[1] XML at World Wide Web (WWW) Consortium. http://www.w3.org/xml

[2] Hewlett-Packard Company. E-Speak Architecture Specification. Version Beta2.2. December 1999. http://www.e-speak.net/library/pdfs/E-speakArch.pdf

[3] D. Rogers. *BizTalk service framework*. Microsoft Corporation. http://www.biztalk.org

[4] Object Management Group. *The common object request broker: Architecture and specification*. Revision 2.0, July 1995 http://www.omg.org

[5] A. Dan and F. Parr. *An Object implementation of network centric business service application (NCBSAs): conversational service transactions, service monitor, and an application style*. OOPSLA’97, Business Object Workshop III.

[6] J. T. Park and J. W. Baek. *Web-based Internet/Intranet service management with QoS support*. IEICE Trans. Commun., e82-b:11, 1999.

[7] ARM Working Group. Application Response Measurement API Guide. 1997. http://www.omg.org/regions/cmgarmw/index.html

[8] K. Evans, J. Klein, and J. Lyon. *Transaction Internet Protocol – Requirements and Supplemental Information*. 1998. http://www.landfield.com/rfcs/rfc2372.html

[9] A. Sahai, V. Machiraju, and K. Wurster. Managing Next Generation E-Services. HPL Technical Report 2000-120. http://lib.hpl.hp.com/techpubs/2000/HPL-2000-120.html.

[10] A. Sahai, J. Ouyang, V. Machiraju, and K. Wurster. End-to-End E-service Transaction and Conversation Management through Distributed Correlation. HPL Technical Report 2000-145. http://lib.hpl.hp.com/techpubs/2000/HPL-2000-145.html

[11] Web Logic (J2EE implementation) from BEA. http://www.bea.com